Fixed Time, Tiered Memory, and Superlinear Speedup

John L. Gustafson
Ames Laboratory-USDOE
Ames, IA 50011

Abstract

In the problem size-ensemble size plane, fixed-sized and scaled-sized paradigms have been the subsets of primary interest to the parallel processing community. A problem with the newer scaled-sized model is that execution time increases for problems where operation complexity grows faster than storage complexity. The *fixed time model* is introduced, which, unlike the scaled model, implies the need to reduce problem size per processor. This reduction causes *uniprocessor* speed to vary. Historical ensemble models hold uniprocessor performance flat as problem size varies, even beyond physical memory size. However, tiered memory can make performance **increase** instead of decrease as problem size per processor shrinks, and workload can shift to routines with higher speed as the problem is scaled. **Superlinear speedup** results in such cases. Superlinear speedup, far from being an anomaly, becomes commonplace when the performance model makes realistic assumptions about memory speed and problem scaling.

Historical Background: Superlinear Speedup

Enough has been written on the subject of superlinear speedup to merit a survey article on the subject [5]. The initial counter-reaction to the notion of superlinear speedup goes something like this: "For a P-processor algorithm, simply execute the work of each processor on a single processor, and the time will obviously be no worse than P times greater. It will usually be *less*, because sources of parallel inefficiency are eliminated." Faber *et al*. [1] have used this argument as a "proof" of the impossibility of superlinear speedup. (The proof assumes fixed problem size, and that a single processor has all hardware necessary to duplicate the needs of the parallel algorithm.)

A counter to this has been supplied by D. Parkinson [6]. Parkinson's idea is that the serial processor has "loop overhead" in executing something P times, whereas the P-processor computer does not, permitting it to be more than P times faster. (The idea assumes a loop is necessary to do something P times on a serial computer, clearly not true if program memory can store the straight-line code.)

Besides the latter effect, Helmbold and McDowell's survey [5] mentions other apparent sources of superlinear speedup: hidden memory latency, subdivision of system overhead, and randomized algorithms. In the last case, independent processors traverse a solution space with better luck or less context switching cost than a single processor. Generally, historic explanations of superlinear speedup have turned out to be inefficiencies in the serial version caused by a sub-optimal program or by insufficient uniprocessor hardware.

Superlinear speedup that results from inefficiency in the serial algorithm is ephemeral and not particularly interesting. The sections that follow point out two new sources of superlinear speedup: The first is the different speeds of memory inherent in distributed memory ensembles. (Although it resembles the insufficient uniprocessor hardware argument, the effect is too fundamental to distributed memory design to dismiss). The second is the shift in time fraction spent on different-speed tasks, for which I present preliminary experimental evidence.

Historical Background: Performance Models

Traditional parallel computer performance evaluation has fixed problem size and varied the number of processors, the so-called *fixed-size model*. The necessity of the resulting serial bottleneck was refuted at least as early as [7], which describes scaling the problem to increase parallel content. Reference [3] developed the *scaled-size model,* and [4] substantiated it by experiments on a 1024-processor hypercube. The scaled size model specifies that the storage complexity grows in proportion to the number of processors.

A third model is the *fixed-time model*, in which the problem is scaled to take a constant time as processors are added. Worley [8] explored this model for partial differential equation problems, and references [3] and [4] state it as the preferred model. However, the 1000+ speedups described in [4] were for the scaled model, and were based on a hypothetical uniprocessor capable of running the entire 1024-processor job.

Speedup and Efficiency Definitions

Almost every paper on parallel speedup makes the following definition: "Speedup is the ratio of the uniprocessor execution time to the execution time on the parallel processor." *Speedup* is the ratio of speeds, not times. *Speed* is work divided by time. Work can be defined as essential floating point operations, instructions, memory references, or whatever seems a reasonable currency on a given system. The choice of definition for *work* does not affect the arguments presented here.

$$Speedup = \frac{\left(\frac{\text{Parallel work}}{\text{Parallel time}}\right)}{\left(\frac{\text{Uniprocessor work}}{\text{Uniprocessor time}}\right)} \qquad (1)$$

The fixed-time model assumes work is constant, resulting in simplification to the ratio of times. Since problems generally scale to fit the time that a user will tolerate, **we avoid this simplification**. If anything is constant in practical computer use, it is the *time*. Hence, one might simplify speedup as the ratio of parallel work to uniprocessor work done in a given amount of time. Definition (1) is the one used in this paper.

Efficiency is traditionally defined as speedup divided by the number of processors. The definition assumes the impossibility of superlinear speedup, guaranteeing that efficiency cannot exceed unity. Since the arguments presented below show that the quantity often exceeds unity, the term "Efficiency" is a misnomer in this context.

A Fixed-Time Example

On ensemble computers, simply replicating the problem on every processor will usually make total execution time increase by more than just the cost of parallelism. Fixing work per processor instead of storage per processor keeps run time nearly constant. A simple example is that of matrix factoring.

Consider the simple problem of solving N equations in N unknowns, with full coupling between equations (dense matrix representation). Arithmetic work varies as N^3, with storage varying as N^2. On a P-processor distributed memory system, simply replicating the storage structures on every processor will not generally lead to a fixed run time, since the arithmetic work to solve a matrix with PN^2 elements is $P^{3/2}N^3$, whereas a fixed time with little parallel overhead on P processors would call for PN^3 arithmetic work. This means that the scaled model execution time increases as $P^{1/2}$.

This situation appeared in the Wave Mechanics, Fluid Dynamics, and Structural Analysis problems run at Sandia [4], which similarly involved order N^2 data storage and arithmetic complexity of order N^3. On the 1024-processor hypercube, to simulate a like amount of physical time (or convergence accuracy for the structural analysis problem) took about $1024^{1/2} = 32$ times as much ensemble computing time. It was then that we realized that the historical "Just make the problem larger!" argument for distributed memory computing might be simplistic to the point of being fallacious. The scaled model is still the best one to use if storage rather than time dictates the size of the problem that can be run.

For these "N^2 - N^3 " problems, it is useful to think about increasing the ensemble size by powers of 64. (There is now a commercial MIMD hypercube offered with $64^2 = 4096$ processors). With 64 times as much computing power, increasing N by a factor of 4 increases the work by a factor of $4^3 = 64$, which should keep execution time about constant if parallel overhead is low. However, the total data storage then only increases by a factor of $4^2 = 16$, not 64. Thus, each processor actually decreases in local storage requirements by a factor of 4. With a typical distributed memory approach of using n by n domains on each processor, the domains shrink to $n/2$ by $n/2$ for every factor of 64 increase in the number of processors. *Fixed-time performance models must shrink the subdomains as the number of processors P increases, if work grows faster than storage*. For the N^2 - N^3 problems, the linear size n of a subdomain will vary as $P^{-1/6}$ if we assume linear performance increases. (This is the reason for picking powers of 64 for P in this example). On a log-log graph of problem size and ensemble size, the ideal fixed-time model appears as a line of slope 2/3, the ratio of the exponents for storage complexity and space complexity (see Figure 1).

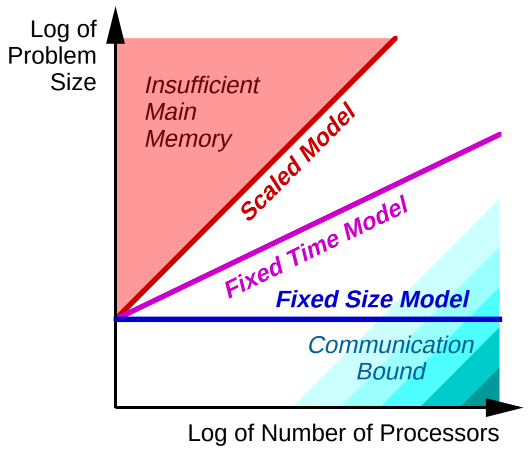

Figure 1. Problem Size vs. Ensemble Size

This example, like most performance models, assumes that uniprocessor performance is constant as problem size is varied. This assumption becomes increasingly implausible when dealing with massively parallel ensembles. A refinement is given in the next section.

Uniprocessor Performance as a Function of Problem Size

In the fixed-size model, data storage per processor decreases rapidly as the number of processors increases. In a distributed memory computer, this effectively decreases the problem solved by each processor. The *scaled size* performance model used in much of the Sandia work eliminates this problem, but at the cost of increased execution time. From the preceding section, we observe that problem "size" (in the sense of storage) must shrink as the number of processors increases, although more gradually than for the fixed size model. (The fixed time model implies a limit to parallel speedup, since one cannot divide problems more finely than one variable per processor. Fortunately, the fixed-time limit is several orders of magnitude greater than the limit implied by the fixed-size model.)

Uniprocessor performance, in effective MFLOPS or any other unit, depends on problem size. Traditional performance analysis has tended to ignore this, or focus on only minor effects within a narrow range of problem sizes. One might assume performance always increases with problem size, because of the amortization of fixed overhead (loop overhead, initialization, vector startup, etc.) Therefore, the decrease in uniprocessor performance appears as another source of parallel inefficiency as a problem is "spread out" among processors.

We define $U(n)$ to be the uniprocessor performance as a function of its problem size n, ignoring the cost of cooperation with other processors in an ensemble. If the cost of cooperation is small, then a P processor system might provide performance given by $P \bullet U(n/P)$ for a fixed problem, $P \bullet U(n)$ for a scaled problem, and something intermediate for a fixed-time problem, like $P \bullet U(n/P^{1/6})$.

The Sandia experiments showed $U(n)$ to be an increasing function of n for the range of n studied [4], like that shown in Figure 2.

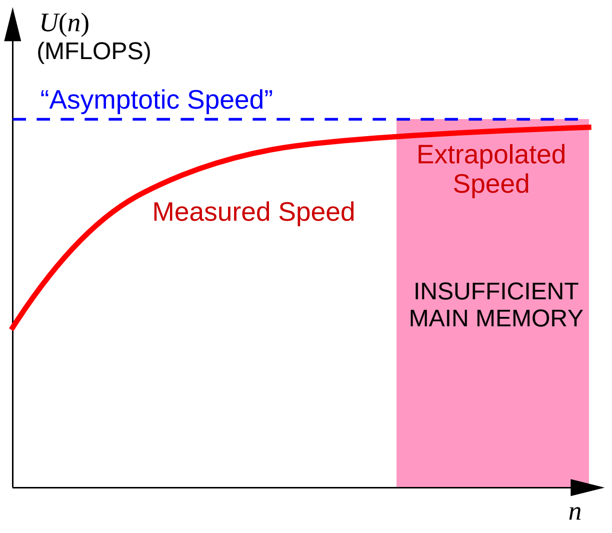

Figure 2. Small-Scale Uniprocessor Speed vs. Problem Size

The reason was that n was restricted to that range for which memory could be regarded as "flat," that is, equal speed independent of size or location. It was easy to calculate the speed of a hypothetical single processor capable of running the ensemble problem at the asymptotic speed, even though that would require a thousand times as much memory as was physically available.

Although appealing from the viewpoint of being a controlled quantity amid a swamp of variables in computer performance evaluation, the assumption that $U(n)$ is constant is not realistic generally. Because of tiered memory effects, it is possible for $U(n)$ to *decrease* as n increases.

Tiered Memory and Superlinear Speedup

Every modern computer has more than one memory speed, if we define “memory” in a very general way. Registers, data caches, RAM, disks, and even tape drives are considered “memory” here. Also, it is almost universal that the size of a memory type increases as the speed of the memory type decreases. Distributed memory computers are no exception to this rule. The term “tiered memory” is sometimes used to describe different speed versions of RAM, but here we use it more generally. Each processor in an ensemble has registers and local RAM; some ensembles have distributed mass storage facilities as well.

When distributed memory problems grow beyond a convenient fit into available RAM, that part of the performance plane is labeled “insufficient memory” and is not explored (Figure 1). Yet, there is mass storage which *could* hold the problem, at the cost of one or two orders of magnitude speed decrease. In the other direction, when a problem shrinks to only a few variables per processor, we could use the registers or other small, fast memory instead of RAM. Both choices call for reprogramming in most environments, but both would realistically be undertaken if a practical situation required it. The performance plane could be redrawn as shown in Figure 3.

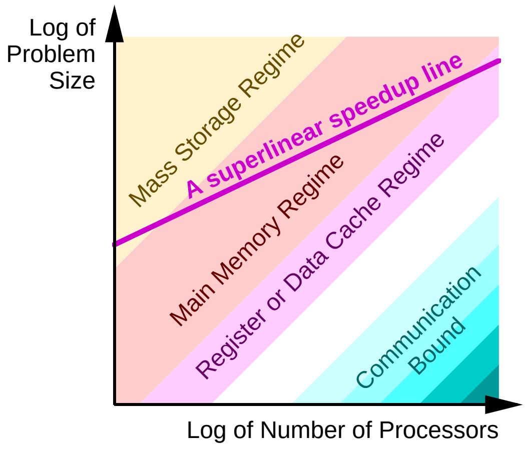
Log of Problem Size
Mass Storage Regime
A superlinear speedup line
Main Memory Regime
Register or Data Cache Regime
Communication Bound
Log of Number of Processors

Figure 3. Revised Performance Plane

Any performance model with a line in the plane of slope less than 45˚ (the scaled model) will result in a *U*(*n*) function of the following general form:

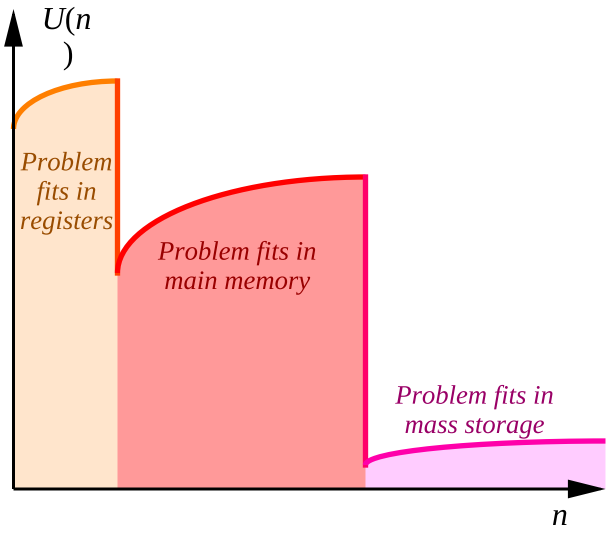

Figure 4. Large-Scale Uniprocessor Speed vs. Problem Size

Superlinear speedup can result whenever problem size per processor is reduced, whether from fixed sized or fixed time performance evaluation, **such that *U*(*n*) crosses regimes and appears decreasing instead of increasing**. The decrease must be enough that the usual sources of parallel inefficiency (load imbalance, serial algorithm steps, interprocessor communication) are compensated.

The effect can be subtle when a processor has a *data cache*. The regime is entered in a manner transparent to the programmer; this effect has already been observed on shared memory computers [5]. A typical case of superlinear speedup is that observed on the Alliant FX/8 when a problem spread out among all eight of its processors now fits into the cache, and achieves more than eightfold speedup. This happens, for example, on large linear algebra problems for which the multiply-add operations consume more bandwidth than the main memory can deliver. Hence, the reasoning presented here is not restricted to distributed memory machines. Interestingly, most people react to this form of superlinear speedup as anomalous, like an improperly run experiment. It is not anomalous. It is as inescapable as the laws of physics that processor-memory speeds increase as a problem is spread out over many processors.

Distributed memory computers force us to recognize these issues because they scale well to large numbers of processors. In benchmarking MIMD systems of over 1000 processors, the large dynamic range of problem sizes forces us to abandon the simplification that memory is “flat.” In recognizing tiered memory, we have the cheerful discovery that “efficiency” needs *not* decrease as we add processors. Superlinear parallel speedup, far from being the result of “inefficient” serial execution, becomes inescapable when the performance model makes realistic assumptions about the speed of memory and the way problems scale.

Superlinear Speedup from Changing Routine Profile

Another type of superlinear speedup results when problem scaling causes *more time to be spent in faster routines*. To use a down-to-earth example, suppose the task is to move an upright piano, the work measure is distance moved, and the fixed time is 30 minutes. A single person might succeed in moving the piano a few feet, perhaps getting it out the door, while a truck sits idling. Two people might lift the piano, load it onto the truck, and drive it twenty miles down the freeway. The speedup of using two people instead of one would be over a thousand times, because a larger fraction of the time is spent at high speed.

Consider the matrix factoring problem, this time *together with the problem of setting up the matrix*. The setup will take order n^2 work and the factoring will take order n^3 work. For small problems, setup might dominate the work, depending on the cost per matrix entry. The factoring approaches 100% of the work as n increases. Both steps can readily be done in parallel. In the fixed-time model, the fraction of the time spent on factoring increases with the number of processors. If the factoring proceeds at a higher MFLOPS rate than the setup (often the case) *then each processor will run faster (more operations per second) as the result of using more processors*. This effect is independent of the tiered memory effect.

This reasoning is the theory of superlinear speedup by shifting algorithm profile. To test it experimentally, I chose a real application from the scientific literature, and with collaborators D. Rover and S. Elbert at Ames Lab created a program for it that scales readily. We also created a distributed memory parallel version of the program, and optimized both serial and parallel versions to the limit of our abilities (including the use of assembly language).

The application is that of computing the radiosity on the interior of a box for realistic scene rendering, as described in [2]. Time for the program was fixed at one minute, and elapsed time was measured for the *entire* application: input of the geometry description from disk, setup of the matrix representing the equations, solving the equations, and storing the solution on disk. Three months were spent reducing the serial version to an efficient form, to reduce the likelihood of superlinear speedup from spurious sources. The serial version was then parallelized for a small NCUBE system. The speed in MFLOPS, as a function of *P*, was measured as follows:

Table 1
Speedup on Radiosity Application

P	Problem Size, N	MFLOPS	Speedup
1	112	0.067	1.00x
2	150	0.138	2.06x
4	200	0.279	4.16x

Even after extensive use of assembly language the problem setup, with intrinsic functions and irregular sequences of operations, ran at 0.06 MFLOPS per processor. The matrix solution, however, ran at 0.12 MFLOPS for large *N*. On one processor, problem setup took 60% of the time, so the speed was close to 0.06 MFLOPS. On four processors, the larger *N* possible in a one-minute run make factorization take most of the time, so the speed *per processor* increased to about 0.07 MFLOPS. The effect would have been more dramatic except for the lack of parallelism in the input, output, and backsolving tasks. With further work, these will also run in parallel and the superlinearity should approach about 6x speedup on 4 processors.

Figure 5 illustrates the effect described:

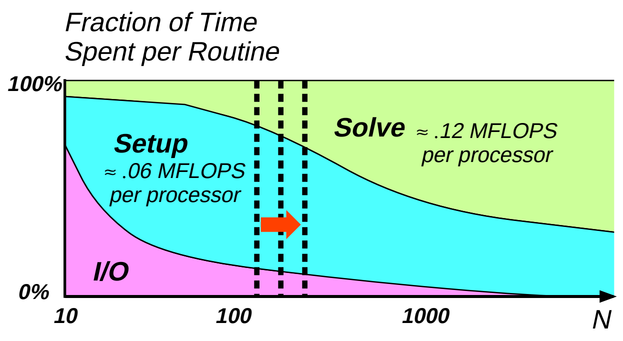

Figure 5. Routine Fraction vs Problem Size

The time complexities are $O(N)$ for input/output, $O(N^2)$ for setup, and $O(N^3)$ for the solve. This seems typical for scientific problems. It also seems typical that the solve or other kernel that dominates for large *N* also has the highest MFLOPS rate, with regular, hazard-free operation sequences.

A Fixed-Time Paradox

A curious fact emerges when one examines the speedups of *individual routines*. For example, in going from one to four processors, the setup speedup was 3.9, the solve speedup was 3.7, and the remainder actually slowed down: 0.7 (because of parallel overhead and unparallelized tasks). **Yet, the overall speedup is greater than 4**.

This counterintuitive result shows that component speedups are not additive, even with appropriate weights. To go back to the piano-moving example, the two men might be only 1.9 times faster at moving the piano to the truck than a single man, and the truck is certainly 1.0 times faster for having a second passenger, yet the overall speedup of the task in terms of miles moved in a given time can be enormous. Examples like these show why “efficiency” should not be defined as speedup divided by the number of processors!

SUMMARY

We have shown theoretical and experimental evidence for non-spurious superlinear speedup. Specifically, speed per processor is *not* constant; it changes with the size of the global problem because of distance to memory and changing algorithm complexity. Work done by a distributed memory ensemble is *not* equivalent to the work done by a single member of that ensemble running the distributed tasks sequentially. Speedup is the ratio of speeds, *not* times. Collectively, these assumptions give reason to expect that an ensemble computer will often be “more than the sum of its parts” in performance on problems of practical interest.

This work is supported by the Applied Mathematical Sciences Program of the Ames Laboratory-USDOE under contract No. W-7405-ENG-82.

References

[1] Faber, V., Lubeck, O. M., and White, A. B., “Superlinear Speedup of an Efficient Sequential Algorithm is Not Possible,” *Parallel Computing*, **3** (1986), pp. 259-260.

[2] Goral, G. M., Torrance, K. E., Greenberg, D. P., and Battaile, B., “Modeling the Interaction of Light Between Diffuse Surfaces,” *ACM Computer Graphics*, Volume 18, Number 3 (1984), pp. 213-222.

[3] Gustafson, J. L., “Reevaluating Amdahl’s Law,“ *Communications of the ACM,* Volume 31 (1987), pp. 532–533.

[4] Gustafson, J. L., Montry, G. R., and Benner, R. E., “Development of Parallel Methods for a 1024-Processor Hypercube,” *SIAM Journal on Scientific and Statistical Computing*, Volume 9 (1988), Number 4, pp. 609–638.

[5] Helmbold, D. P. and McDowell, C. E., “Modeling Speedup(n) greater than n,” *1989 International Conference on Parallel Processing Proceedings*, (1989), Volume III, pp. 219-225.

[6] Parkinson, D., “Parallel Efficiency can be Greater than Unity,” *Parallel Computing*, **3** (1986), pp. 261-262.

[7] Seitz, C. L., “The Cosmic Cube,” *Communications of the ACM*, Volume 28 (1985), pp. 22-33.

[8] Worley, P. H., “The Effect of Time Constraints on Scaled Speedup,” *Report ORNL/TM-11031*, Oak Ridge National Laboratory, January, 1989.